An Automated Test Strategy Based on UML Diagrams

F. Basanieri *, A. Bertolino *, E. Marchetti *, A. Ribolini *, G. Lombardi **, G. Nucera **

*Istituto di Elaborazione dell'Informazione
CNR, Pisa, Italy

** ERICSSON LAB ITALY

f.basanieri@iei.pi.cnr.it
bertolino@iei.pi.cnr.it
e.marchetti@iei.pi.cnr.it
a.ribolini@iei.pi.cnr.it
gaetano.lombardi@tei.ericsson.se
giovanni.nucera@tei.ericsson.se

Abstract

This is a work-in-progress report about the Cow_Suite tool currently under development for automating CoWTeSt (COst Weighted TEst STrategy), an original strategy for selecting and prioritarising test cases. The tool supports managers to schedule and make cost estimates of the integration test stages since the early phases of development. The derivation of test cases is based on the software analysis and design documentation, and uses the UML-based original test methodology UIT, Use Interaction Test. We describe the tool architecture and show the provided features through an example of application to a real case study with some results.

CowSuite tool and CoWTeSt have been planned and developed by PISATEL. PISATEL is a Software Laboratory established in Istituto Elaborazione Informatica (IEI) Pisa in cooperation with Ericsson Lab Italy (ERI). Starting in January 2001, teams from IEI and ERI are conducting joint applied research on the Software Engineering, and its application to telecommunications sector. The hope is that, on the one hand, these projects will make evident to the research community the complexity of the activities and the real world problems and constraints faced by the industrial partner while, on the other, they will provide an effective benchmark for the validation and refinement of the latest research results from the academic labs.

1. Introduction

In this paper we report about the current state of an on-going research project aimed at developing methods and tools for automated testing based on UML [5], [6]. The two main requirements of our project are the following:

- test planning and management should use the UML diagrams developed during specification and design, but should not impose to the UML designers any additional formalism, or ad-hoc effort specifically for testing purposes.
- the test cases should be derived in incremental, systematic way and in manageable numbers in line with project costs and schedules.

Several emerging research proposals for UML based test methods, e.g. [3] and [4], require extensive additional effort or even additional formalisms from UML designers: we believe that such proposals can hardly find wide industrial acceptance. On the contrary, our goal is to use the existing design diagrams and even to have tools compatible with currently available tools for UML design, so that the proposed test methodology can be adopted by industries using UML with very little additional effort. Of course, one problem to handle here will be the inherent subjectivity and potential ambiguity of UML diagrams.

With regard to the second requirement, we believe that, again for pushing industrial acceptance, any test methodology should incorporate considerations of testing cost and project schedule. For

very complex applications, such as modern distributed systems, the testing stage can be very expensive, and very difficult to manage in incremental way. We will rely on the UML design to guide an incremental test strategy: of course, we can do this to the extent that the designers apply rigorous and formalised design methodologies.

In response to the above requirements, we have developed a test methodology that consists of two main components: a method to derive the test cases, called UIT (Use Interaction Test), already presented in [1], and a test management strategy, called Cowtest (Cost Weighted Test Strategy), presented in [2], to help decide which and how many, among the derived test cases, should be launched. Cowtest and UIT are currently being implemented in a unified test environment, called Cow_Suite (Cow pluS UIT Environment), whose architecture is presented here.

In particular, it is possible to apply Cow_Suite in two ways: if a certain resource investment has been fixed for a testing phase, it is possible to estimate how many test cases to execute, but more importantly to prioritise them and pick those that are judged more useful to assure quality, stability and to evidence problems of the developed product. On the other hand, if the testing stopping rule is given by the coverage of a fixed percentage of functionalities (e.g., 90% functional coverage), Cow_Suite helps to evaluate from the very first stages the cost of such a target and consequently to choose the most suitable test cases, as for the previous case.

Cow_Suite uses the UML diagrams already developed for the analysis and design phases. To derive the test cases the tool uses the UML-based UIT methodology. Since Cow_Suite has been designed to be compatible with the Rational Rose environment [7], in particular using REI (Rational Rose Extensibility Interface), it can be easily adopted by the many industries already using this package, with little additional effort.

The paper is organized as follows: in Section 2 we give the basic knowledge required for the application of the UIT methodology. Then in Section 3 we describe the details of the Cowtest strategy and present in Section 4 and 5, respectively, the tool architecture and a case study.

2. Use Interaction Testing

The method implemented in Cow_Suite for test derivation is UIT. We report here only a brief summary of it, remanding to [1] for the complete description. The UIT method derives integration test cases, at different integration or abstraction levels, from UML design diagrams.

In Figure 1 we show an activity diagram describing the steps necessary for test derivation according to UIT. The first step consists in finding all the Use Case Diagram (UCDs) and Use Cases (UCs).

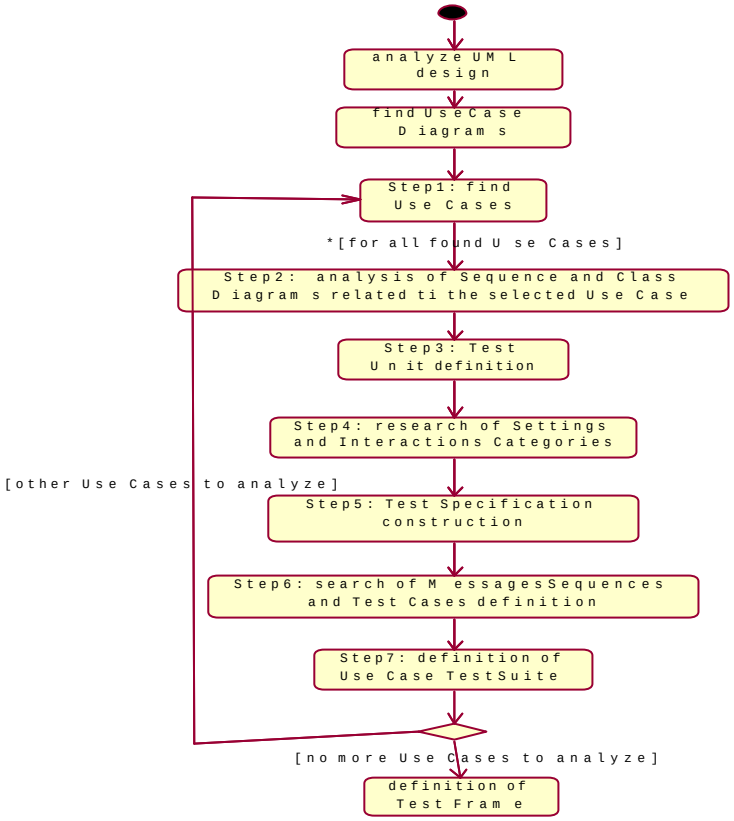

Figure 1: Activity diagram for test derivation

Then the Sequence Diagrams (SDs) and the Class Diagrams (CDs) are analysed (Step2), identifying the messages that the objects in the SDs exchange with each other, and the relative parameters. Now, each object inside a SD is considered as a Test Unit, in the sense that it can be separately tested and represents a possible use of system (Step3). For each of these Test Units, the relevant Settings and Interactions Categories are derived that are, respectively, environment parameters (or state variables) and messages coming from other Test Units (Step4). In this phase to capture information about parameters values and messages definition we also use the information of the related Class Diagrams.

A Test Specification is derived finding for each identified category of a Test Unit all the possible values and constraints (Step 5). Then, observing the temporal ordering of SD messages, it is possible to find the Message Sequences, i.e., a set of messages exploited by objects to define and elaborate specific functionalities. Inside each Message Sequence, we can find a set of Interactions Categories (messages of this sequence) and Settings Categories (attributes that affect the messages) (Step6). Finally, an executable Test Case is constructed from a Test Specification, taking each of every possible choice, for each involved category (Step7).

We report below an example of some test cases, generated for the example described in Section 5. Note that the description of the test cases at this level actually remains abstract and can be considered as the specification of test classes. The real executable test cases will be derived instantiating the involved categories values.

TEST CASE 7

Description:
PreCondition:
Test Case 6
Flow of event:
AccessAgent->getAccessType()
[Post: checkCLIP(SetupMsg, caller, AccessType)
and CallCase elaboration from Li+i.TestCase 4]
Categories:
SettingsCategories:
SetupMsg =
AccessType =
InteractionsCategories:
getAccessType()
PostCondition:
Comment:

TEST CASE 8.1

Description:

PreCondition:

Test Case 7

Flow of event:

[if pEnterprise!=NULL]
NonStdSetup->setEnterprise
routingResult =doLRQ(caller, callee, Enterprise, callcase,
AccessAgent, BGAResult)

Categories:

SettingsCategories:

PEnterprise =
Caller =
Callee =
Callcase =
AccessAgent =
BGAResult =

InteractionsCategories:

setEnterprise =
doLRQ =

PostCondition :

Comment :

TEST CASE 8.2

Description:

PreCondition:

Test Case 7

Flow of event:

getEnterpriseForSourceAddress(caller)
[else]
routingResult =doLRQ(caller, callee, Enterprise, callcase,
AccessAgent, BGAResult)

Categories:

SettingsCategories:

PEnterprise =
Caller =
Callee =
Callcase =
AccessAgent =
BGAResult =

InteractionsCategories:

getEnterpriseForSourceAddress =
doLRQ =

PostCondition:

Comment:

TEST CASE 8.3

Description:

PreCondition:

Test Case 7

Flow of event:

[if RoutingResult=NULL or BGAResult = NULL]
[get release reason release(Handler, reason) from Li+1.TC]

Categories:

SettingsCategories

Handler =
Reason =

InteractionsCategories

PostCondition:

Comment :

3. Cowtest strategy

In this section we summarise the strategy that will be implemented by the Cow_Suite for selecting test cases according to various possible industrial needs. The complete description of how to derive the basic structure (a weighted tree) used for test derivation and the test case selection in view of different project exigencies is in [2].

Starting from the main UCD, describing the system functionalities at a very high level, the UCDs are organized in a hierarchical tree. Then to each Use Case we associate the relative SDs which describe the objects interactions, and exchanged messages, used to realize the Use Case scenarios. Finally from each SD the test cases are derived using the UIT method. In Figure 2 we show an example of a tree relative to the case study that will be presented in Section 3.

After the tree construction, it is necessary to label every node with a value representing in a sense the "importance" of this node (be it a UC, or a SD scenario) with respect to the other nodes at the same level in the tree. These values, called weights, must belong to the [0,1] interval and must be assigned (by the tester) in such a manner that the sum of the weights associated to all the children of one node is equal to 1.

The weight should be as high as critical is the functionality represented by the associated node (UC or SD). These values contribute to define a relative importance factor, in terms of how risky is that node and how much effort should we put to test it, for each element belonging to the integration stage considered.

The last step of Cowtest strategy is to calculate for every node its real final weight, i.e. the product of all the nodes weights on the complete path from the root to this node. The final weight associated with each leaf of the tree becomes therefore an element of discrimination for choosing amongst the tests to execute and will be used in two different manners.

The first is the case in which a certain test budget is available, or a fixed number of test cases must be executed. In such a case, Cow_Suite selects the most suitable distribution of the test cases among the functionalities developed on the basis of the leaves weights and with respect to the available budget.

The second situation considered is that a certain percentage of functionalities must be covered (e.g. 90%). In this case the tool can drive the functional choice, highlighting the most critical system functionalities and properly distributing the test cases.

4. Cow_Suite Architecture

Cow_Suite is developed using the *Rational Rose Extensibility Interface (REI)*[1]. The REI object (available with the Rational Rose package) can be included directly in Visual Basic or C++ projects using the Microsoft OLE (Object Linked and Embedded) interface system [8].

The tool is composed by six different modules:

1 MDL Analyser
2 Test Tree Generator
3 Weights Manager
4 Test Case Generator
5 Test Case Builder
6 Test Case Checker

[1] The tool works in Microsoft environment (Windows 95/98/NT/2000) and is being developed using Microsoft Visual Studio (version 6) with Service Pack 4.0. Moreover Cow_Suite requires the Visual Basic RunTime (version 6 with Service Pack 4.0) and Rational OLE server available as Type Library.
The platform requirements of the tool are:
- Microsoft Windows 95/98/NT/2000
- minimum Pentium 150MHz or fasters CPU
- minimum 64Mbyte of RAM (recommended 128Mb RAM)
- minimum 65Mb of disk space
- Rational Rose or server REI (by TLB) installed

MDL Analyser: This module analyses the Rose description of the project model (given in a .mdl file) deriving the following components: Classes (with their attributes and methods), Actors, UseCases, Sequence Diagrams, Package Diagrams /Class Diagrams, Activity Diagram. In this manner, the information necessary to apply the test selection strategy is extracted and passed as an input to the next module.

Test Tree Generator: using the information derived by the MDL Analyser, all the UCDs and SDs are organized in a hierarchical tree. The tool associates to each node: a level identification number, representing the position of the node in the tree, and a default weight such that the sum of this weight plus the weights associated to all its brothers is equal to 1. The identification number is visualized below each tree level. The weight of each node is visualized in a box positioned before the node-name (see Figure 2).

In detail the tree structure is derived considering the explicit links (i.e. the link between the OpenSpecification and the diagram) or using the associations (with or without stereotypes) and the relations between the classes.

Weights Manager: this module interacts with the user for assigning the real weight to each node and choosing the criterion for test cases selection.

The first kind of interaction can be driven in two different ways: selecting directly a node on the tree or a level number. In the former the user can modify directly on the visualized tree the node's weight using the form associated to the node. In the latter only the chosen level of the tree is entirely visualized and the user can insert the proper values for each node. Checking that the sum of the user assigned weights of a level nodes is equal to 1 is also a task of this module.

The second user interaction is useful to determine the criterion to be used for test selection: fixing the maximum number of test cases to be executed or the percentage of functionalities to be covered.

According to the chosen selection criterion the proper set of test cases can be derived.

Test Case Generator: the tasks of this module are:

a) To query the user for the deepest integration level he/she is interested in and consequently calculate the final weight of every leaf.
b) To implement the UIT method for test generation
c) To associate to each SD its test cases, possibly organizing them in a hierarchical manner.
d) To calculate and visualize the number of test cases for each SD depending on the chosen test case selection criterion.
e) To associate to each SD the frames of the test cases that should be instantiated.

Test Case Builder: this module interacts with the user for test case implementation, asking for the necessary parameters values and checking if the test case has been already instantiated.

In the current state Cow_Suite is able to visualize in a tree structure all the test cases generated for a SD. Inside a generated test case (see Section 2) we can find the list of all its Settings and Interactions categories and their values. Moreover, the user can interact with the tool adding and removing the categories or eventually changing parameters, operations, categories values or even test case structure. The changes involving the UML design are finally saved in a new .mdl file.

Test Case Checker:

The tool maintains information about the test cases generation. This module will realize the comparison of different versions of the same .mdl file. The discovered differences, like, for example, the existence of new test cases or changes in those already generated, are saved into a separate file. The evaluation of the cost and impact required for the updates to the test plan with respect to the "official version" is derived analyzing this file.

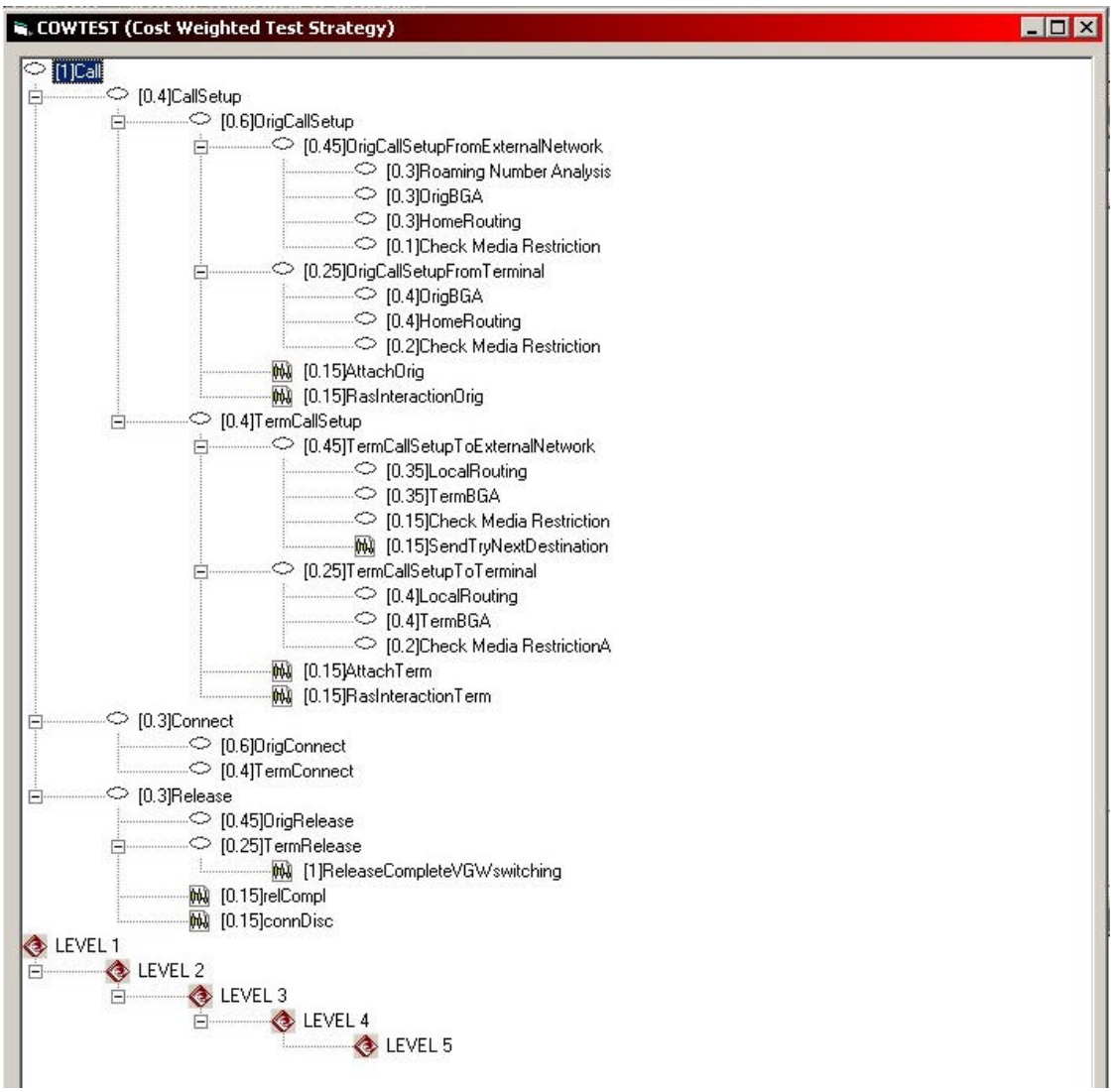

Figure 2: Annotated Cowtest tree

5. Example

We present an example from a case study on which we are applying and developing Cow_Suite. The case study consists of a real project provided by a Telecommunication software developer. We consider here only a part of the analysed system, named the Call Manager.

The Call UC can be divided into tree sub-UCs:

1 CallSetup, that is the phase in which caller and receiver are identified and localized,
2 Connect, in which the partners, after their connection, can communicate with each other;
3 Release, in which one of the users involved in the connection terminates the call;

In Figure 2 we can see the complete tree with all UCs and SDs involved, as developed according to the Cowtest strategy.

A SD describes one of the subsystem functionalities. Each functionality can be realized inside a software component, like modules or packages, or obtained by the interactions of several different components. In this manner the test cases can be developed both to verify the integration between different system parts and to investigate on the specific interactions between implementation objects.

Cow_Suite currently does not yet automatically derives the results showed in the following tables relative of the presented case study. We report these tables for giving an idea of how the tool will behave. As already explained, the leaves weights can be used in different manner depending on the project needs.

Table 1 is a summary of the situation in which the test manager uses Cowtest having a fixed number of test cases to develop, in the example shown 500, and needs to decide on how to distribute these test cases among the specified functionalities. The table is organized in the following manner: the first and the second columns hold respectively the nodes names and the tree levels. The third column shows the node weights as the user inserted them in the tree. In the fourth and fifth columns there are the final weights that Cow_Suite will use to distribute the test cases among the leaves. Precisely, the fourth column reports the final absolute weight of each node, obtained by multiplying the relative weights of all nodes between it and the root. The fifth column only considers the absolute weights for all leaves at the considered nesting level (5th level, in the example). So, for example, considering the SD relCompl (SD_Release_relCompl in the tables) its final weight (0.45) is given by the product of its weight (0.15) times its father's weight (0.3).

In the same manner we calculate the final weight of the SD SendTryNextDestination (SD_TermCallSetupToExternalNetwork_SendTry in the tables) (0.0108=0.15*0.45*0.4*0.4). Finally, the obtained numbers of tests for each leaf are reported in the sixth column.

Nodes name	Tree Levels	Nodes weights	Final Nodes weights	Final Leaves weights	NTtest
Call	1	1	1	0	
CallSetUp	2	0,4	0,4	0	
Connect	2	0,3	0,3	0	
Release	2	0,3	0,3	0	
OrigCallSetup	3	0,6	0,24	0	
TermCallSetup	3	0,4	0,16	0	
OrigConnect	3	0,6	0,18	0,18	90
TermConnect	3	0,4	0,12	0,12	60
OrigRelease	3	0,45	0,135	0,135	67,5
TermRelease	3	0,25	0,075	0	
SD_Release_relCompl	3	0,15	0,045	0,045	22,5
SD_Release_connDisk	3	0,15	0,045	0,045	22,5
OrigCallSetupFromExternalNetwork	4	0,45	0,108	0	
OrigCallSetupFromTerminal	4	0,25	0,06	0	
SD_OrigCallSetup_AttachOrig	4	0,15	0,036	0,036	18
SD_OrigCallSetup_RasinteractionOrig	4	0,15	0,036	0,036	18
TermCallSetupToExternalNetwork	4	0,45	0,072	0	
TermCallSetupToTerminal	4	0,25	0,04	0	
SD_TermCallSetup_AttachTerm	4	0,15	0,024	0,024	12
SD_TermCallSetup_RasinteractionTerm	4	0,15	0,024	0,024	12
SD_TermRelease_ReleaseComplete	4	1	0,075	0,075	37,5
RoamingNumbersAnalysis	5	0,3	0,0324	0,0324	16,2
OrigBGA_OCSFEN	5	0,3	0,0324	0,0324	16,2
HomeRouting_OCSFEN	5	0,3	0,0324	0,0324	16,2
CheckMediaRestriction_OCSFEN	5	0,1	0,0108	0,0108	5,4
OrigBGA_OCSFT	5	0,4	0,024	0,024	12
HomeRouting_OCSFT	5	0,4	0,024	0,024	12
CheckMediaRestriction_OCSFT	5	0,2	0,012	0,012	6
LocalRouting_TCSTEN	5	0,35	0,0252	0,0252	12,6
CheckMediaRestriction_TCSTEN	5	0,15	0,0108	0,0108	5,4
TermBGA_TCSTEN	5	0,35	0,0252	0,0252	12,6
SD_TermCallSetupToExternalNetwork_SendTr	5	0,15	0,0108	0,0108	5,4
LocalRouting_TCSTT	5	0,4	0,016	0,016	8
CheckMediaRestriction_TCSTT	5	0,2	0,008	0,008	4
TermBGA_TCSTT	5	0,4	0,016	0,016	8
Total				1	500

Table 1: Distribution of test case at different integration stages

We report in Table 2 some results obtained for the situation that a functional coverage is fixed. The weights distribution as well as the number of test case (NTest) will be redistributed and assigned by the tool (see [2] for more details).

The table is organized in the following manner: the first five columns have the same meaning of the previous table. The remaining columns are divided in two parts showing, respectively, the normalized weight and the minimum number of tests with respect to the fixed coverage percentage.

Nodes name	Tree Levels	Nodes weights	Final Nodes weights	Final Leaves weights	70%coverage *nwf norm* /MinNTest		80%coverage/ *nwf norm* /MinNTest		90%coverage/ *nwf norm*/MinNTest		100%coverage/ *nwf norm* /MinNTest	
OrigConnect	3	0,6	0,18	0,18	0,2555366	6	0,2196193	7	0,1965924	8	0,18	23
TermConnect	3	0,4	0,12	0,12	0,1703578	4	0,1464129	5	0,1310616	5	0,12	15
OrigRelease	3	0,45	0,135	0,135	0,1916525	4	0,1647145	5	0,1474443	6	0,135	17
SD_TermRelease_ReleaseComplete	4	1	0,075	0,075	0,1064736	2	0,0915081	3	0,0819135	3	0,075	9
SD_Release_relCompl	3	0,15	0,045	0,045	0,0638842	1	0,0549048	2	0,0491481	2	0,045	6
SD_Release_connDisk	3	0,15	0,045	0,045	0,0638842	1	0,0549048	2	0,0491481	2	0,045	6
SD_OrigCallSetup_AttachOrig	4	0,15	0,036	0,036	0,0511073	1	0,0439239	1	0,0393185	2	0,036	5
SD_OrigCallSetup_RasinteractionOrig	4	0,15	0,036	0,036	0,0511073	1	0,0439239	1	0,0393185	2	0,036	5
RoamingNumbersAnalysis	5	0,3	0,0324	0,0324	0,0459966	1	0,0395315	1	0,0353866	1	0,0324	4
origBGA_OCSFEN	5	0,3	0,0324	0,0324	0	0	0,0395315	1	0,0353866	1	0,0324	4
HomeRouting_OCSFEN	5	0,3	0,0324	0,0324	0	0	0,0395315	1	0,0353866	1	0,0324	4
LocalRouting_TCSTEN	5	0,35	0,0252	0,0252	0	0	0,0307467	1	0,0275229	1	0,0252	3
TermBGA_TCSTEN	5	0,35	0,0252	0,0252	0	0	0,0307467	1	0,0275229	1	0,0252	3
SD_TermCallSetup_AttachTern	4	0,15	0,024	0,024	0	0	0	0	0,0262123	1	0,024	3
SD_TermCallSetup_RasinteractionTern	4	0,15	0,024	0,024	0	0	0	0	0,0262123	1	0,024	3
OrigBGA_OCSFT	5	0,4	0,024	0,024	0	0	0	0	0,0262123	1	0,024	3
HomeRouting_OCSFT	5	0,4	0,024	0,024	0	0	0	0	0,0262123	1	0,024	3
LocalRouting_TCSTT	5	0,4	0,016	0,016	0	0	0	0	0	0	0,016	2
TermBGA_TCSTT	5	0,4	0,016	0,016	0	0	0	0	0	0	0,016	2
CheckMediaRestriction_OCSFT	5	0,2	0,012	0,012	0	0	0	0	0	0	0,012	2
CheckMediaRestriction_OCSFEN	5	0,1	0,0108	0,0108	0	0	0	0	0	0	0,0108	1
CheckMediaRestriction_TCSTEN	5	0,15	0,0108	0,0108	0	0	0	0	0	0	0,0108	1
SD_TermCallSetupToExternalNetwork_SendTr	5	0,15	0,0108	0,0108	0	0	0	0	0	0	0,0108	1
CheckMediaRestriction_TCSTT	5	0,2	0,008	0,008	0	0	0	0	0	0	0,008	1
Call	1	1	1	0	0	0	0	0	0	0	0	0
CallSetUp	2	0,4	0,4	0	0	0	0	0	0	0	0	0
Connect	2	0,3	0,3	0	0	0	0	0	0	0	0	0
Release	2	0,3	0,3	0	0	0	0	0	0	0	0	0
OrigCallSetup	3	0,6	0,24	0	0	0	0	0	0	0	0	0
TermCallSetup	3	0,4	0,16	0	0	0	0	0	0	0	0	0
TermRelease	3	0,25	0,075	0	0	0	0	0	0	0	0	0
OrigCallSetupFromExternalNetwork	4	0,45	0,108	0	0	0	0	0	0	0	0	0
OrigCallSetupFromTerminal	4	0,25	0,06	0	0	0	0	0	0	0	0	0
TermCallSetupToExternalNetwork	4	0,45	0,072	0	0	0	0	0	0	0	0	0
TermCallSetupToTerminal	4	0,25	0,04	0	0	0	0	0	0	0	0	0
Total				1	1	22	1	33	1	38	1	125

Table 2: Weights normalization for different coverage percentages

References

[1] Basanieri, F., Bertolino, A., “A Practical Approach to UML-based Derivation of Integration Tests”, in QWE2000 conference proceeding, Bruxelles, November 20-24, 3T.

[2] Basanieri, F., Bertolino, A., Marchetti, E., “CoWTeSt: A Cost Weighed Test Strategy”, accepted for: ESCOM-SCOPE 2001, London, England, 2-4 April 2001.

[3] Hartmann, J., Imoberdof, C., Meisenger, M., "UML-Based Integration Testing", in ISSTA 2000 conference proceeding, Portland, Oregon, 22-25 August 2000.

[4] Offutt, J., Abdurazik, A., "Generating Test from UML Specifications", in UML conference proceeding, Fort Collins, CO, October 1999.

[5] Rumbaugh J., Jacobson I., Booch J. “The Unified Modeling Language Reference Manual”, Addison Wesley, 1999.

[6] UML Documentation version 1.3 Web Site. On-line at http://www.rational.com/uml/resources/documentation/index.jsps/

[7] http://www.rational.com/products/rose/index.jsps

[8] http://msdn.microsoft.com